École Polytechnique Fédérale de Lausanne

Making BFT State Transfer Tolerate Byzantine Faults

by Andrej Milićević

Master Thesis

Approved by the Examining Committee:

Dr. Gauthier Voron
Thesis Advisor

Dr. Dragos-Adrian Seredinschi
External Expert

Prof. Dr. Rachid Guerraoui
Thesis Supervisor

EPFL IC DCL
INR 311 (Bâtiment INR)
Station 14
CH-1015 Lausanne

January 30, 2025

Acknowledgments

I am thankful to my advisor, Dr. Gauthier Voron, for the guidance, advice, and support he provided throughout this project, and my supervisor, Dr. Rachid Guerraoui, for the opportunity to do my thesis and research at the Distributed Computing Laboratory at EPFL.

I would also like to thank my friends and family, especially my mom, dad, brother, and aunt, for their unconditional love and support throughout the entirety of my Master's studies. None of this would be possible without them.

Lausanne, January 30, 2025 Andrej Milićević

Contents

Abstract

State transfer protocols play an immensely important role in State Machine Replication (SMR), as they are the basis of replicas recovering after inevitable failure and crashes. Besides crashing, in a Byzantine setting a replica can behave arbitrarily, therefore these protocols must be both efficient and very resilient. During these transfers, how a replica uses its limited amount of bandwidth plays a huge role. Up until now, only when requesting the state did replicas follow the dynamic nature of the network, while replicas which provide the data acted like servers. This is inefficient, and leads to an unfair utilization of bandwidth, as we show in this project. Although replicas have to react to changes in the network, they also need to have mechanisms to mitigate the impact of a Byzantine replica. Our approach is to make replicas more reactive with the help of Linux Traffic Control. Replicas keep track of information such as how often others request transfers, and how much bandwidth they consume, thus prioritizing incoming requests accordingly. With our proposed solution, called Adaptive State Transfer (AST), we are able to significantly reduce the time it takes to finish a state transfer, while still being resilient to Byzantine replicas. In the presence of Byzantine replicas, we reduce the transfer time by 28%, bringing it 72% closer to the optimum. In the case where all replicas are honest, we reduce the time by 26%, bringing it 80% closer to optimal performance.

Chapter 1

Introduction

Large-scale infrastructures today need to provide integrity and availability despite their servers crashing for various reasons. This is achieved by replicating the data onto multiple servers, ensuring that it is safe in case some servers fail. Consensus algorithms, such as Paxos [10, 11] and Raft [14], provide a solution to this problem, albeit only in the Crash-Fault tolerant setting. Systems in the real world need algorithms that tolerate Byzantine faults as well. Some research has been done [4, 21] that shows the possibility of implementing and deploying Byzantine-Fault tolerant solutions to this problem, generically called *State Machine Replication (SMR)* [16]. More recently, this has improved significantly with Chop-Chop [3]. Chop-Chop represents a novel take on the Byzantine Atomic Broadcast [6], which achieves line rate efficiency despite its complex protocol. With this improvement, the focus of improving SMR protocols overall has shifted to other aspects of the problem, such as the way the state is logged and stored, or how the state is transferred via the network. In this thesis, we focus primarily on the state transfer part, and specifically how to improve and optimize the bandwidth usage during the state transfer protocol.

Currently, the main problems of transferring the state over the network are, as we see them, the presence of Byzantine replicas, and the large-scale nature of today's systems. Dealing with Byzantine replicas is, as mentioned, nothing new; however, we have to be careful when designing and implementing new algorithms, as we must take Byzantine replicas into account and provide robust solutions. Moreover, as infrastructures and their servers are usually scattered across the world, thanks to public cloud services, latency becomes a big hindrance in the state transfer protocol. Although considering which cloud services to use, or where to place the servers, is important [13], following how the network behaves and changes will be a crucial factor in how the system performs. Recent work [5] shows that replicas which dynamically adjust their requests for the state perform better, reducing the time it takes to complete a state transfer.

There are still potential issues and threats to a replica's performance, even though they adhere to the state of the art research and algorithms. Mainly, nothing prevents a Byzantine replica from

requesting a part of the state over and over again, regardless of whether or not it needs the state. The replica can thus simply keep other replicas busy by always asking for data via the network, therefore slowing down any other recoveries happening at the same time, and degrading the overall performance of the whole system. Moreover, this is not only an issue with Byzantine replicas; an honest replica might just be slow when recovering, or keeps crashing and always needs some piece of the state to fully recover. Therefore, a new solution is needed for any system, not just the ones expecting Byzantine replicas.

In current research, only those replicas which request the data follow the dynamic nature of the network. There is no mention of the replicas which send the data; they just act as a server in that communication and send the requested data when it is needed. Since the mentioned potential problems arise precisely when replicas are constantly asked for the data, it is natural to think of ways how the sender can mitigate these issues. The decrease in performance is inevitable, since replicas have a very finite amount of bandwidth available for the state transfer protocol. Keeping in mind that replicas have to continue their regular operation while the transfer is going on, the amount of bandwidth each replica can allocate for the transfer will be low.

Our approach to solve this issue is to give replicas more control over the bandwidth they have at their disposal. When accepting requests, a replica will still answer with as much speed as it can. However, each replica will keep track of how much bandwidth other replicas previously used. This way, any replica that is noisy and "annoying" will have less priority for its future requests, thus receiving the data at a lower pace, i.e. the sender uses less bandwidth for that particular connection. In order to manipulate traffic in this way, we need to use *Linux Traffic Control (tc)* [1, 9, 19], a subsystem that offers tools for managing network traffic in various ways. We need precise control of the packet-scheduling, therefore we need to apply any and all changes on the low, OS-level. Delegating this work to the operating system gives us kernel-level efficiency and exact bandwidth control, while reducing the complexity of our application logic, since we avoid re-implementing any details surrounding this functionality. Using these tools, and the information gathered from state transfers during the system's lifetime, a replica is able to keep track of the behaviour of all other replicas, and manipulate network packets accordingly. Nothing particular stops Byzantine replicas from still being "annoying" and trying to disrupt others. However, over time, their influence will fade, since their priority will gradually drop from the point of view of other replicas.

With the use of `tc`, and a simple way of tracking each replica's behaviour, we are able to reduce the time it takes to transfer state by 28% in the presence of Byzantine replicas. There was room for improvement between the theoretically fastest possible time (taking only network limitations into consideration and nothing else), and some naive solutions we had to the main problem. We were able to further close this gap, reducing it by 72% on average. Moreover, in the scenarios where multiple honest replicas are recovering simultaneously, the transfer time is reduced by 26%, and the gap is closed by almost 80%. Our results definitely show a great benefit to controlling network packets during state transfer, thus improving on the file-transfer aspect of the entire protocol. We also do not forget we are in the Byzantine setting, and that replicas can behave arbitrarily. Our

solution takes this into account, being mindful of what Byzantine replicas can do, whether or not they can harm the system, and, if so, how much.

The core contribution of this thesis consists of a couple of points. Primarily, it is the reduction of state transfer time in cases where a lot of replicas request a transfer at the same time. We found a case where performance degradation is highly likely, especially in the Byzantine case, and provided a solution yielding very positive results. On the way to this solution, we also show that Linux `tc` can effectively be used in this setting, which was previously not explored. `tc` and its tools were developed a long time ago, when networks were a couple of orders of magnitude slower than today. Therefore, it was not certain or obvious how frequent and rapid changes in the `tc` configuration will behave, and whether or not this would lead to positive results in the end.

In the following section, we describe the core concepts on which our work is based. In Section 3 we discuss and analyze certain problems which may occur during state transfer protocols. Section 4 presents the solution we propose for solving the issue of the lack of bandwidth control in replicas. Subsequently, we show the results of our evaluation of the proposed algorithm in Section 5. Section 6 talks about existing research related to our project, and, finally, we conclude our work in Section 7.

Chapter 2

Background

2.1 State Machine Replication

Distributed and decentralized systems bring with them some very important characteristics, such as integrity, availability, and fault tolerance. To achieve these, all actors in the system must have the same view of the data stored on them, while being robust enough to seamlessly handle crashes and arbitrary failures. These actors are *state machines*; programs with an internal state which changes based on a series of inputs received. In the deterministic case, each input to the state machine generates a defined output, which means multiple identical state machines provide an identical output to a series of inputs. This is exactly what we want, however things get complicated when we add multiple state machines working together, in the pursuit of better fault tolerance. For example, if and when one machine fails, we have our data saved on another machine, and we do not lose our data. When multiple machines (also called *replicas* in this scenario) work together like this, we get a *replicated state machine*, i.e. we implement a technique called *State Machine Replication (SMR)*. SMR offers all the characteristic needed for a distributed system to work properly; ensuring high availability while being fault tolerant, scalability, and handling data correctly regardless of which machine is actually in communication with the outside world.

There are two different classes of fault tolerant systems we distinguish: *Crash Fault Tolerant (CFT)* and *Byzantine Fault Tolerant (BFT)* systems. In CFT systems, all replicas are honest, and behave as their algorithm says. The only failures that occur in these systems are crashes, which means a replica is down for a certain period of time, until it is rebooted or fixed. Therefore, if the system want to tolerate up to f crashes simultaneously, it needs to have a total number of replicas equal to $2f + 1$. In BFT systems, however, replicas can arbitrarily deviate from the algorithm, and these replicas are then called Byzantine replicas. They can do anything they want with their state, or send any message they want to other replicas. Since their presence can cause more trouble than in CFT systems, the total amount of replicas in the system needs to be $3f + 1$, in order to tolerate up to

f failures at the same time.

One crucial step in SMR being robust and fault tolerant is the *state transfer protocol*. When one replica crashes, or is just slow, it misses some data along the way, i.e. misses some inputs. Since all replicas in the system must have the same state, the one recovering needs to somehow get the part of the state it is missing. The recovering replica achieves this by asking other replicas for the state via the network, and waiting for their responses. Whether it is a CFT or a BFT system, there will always be honest replicas that have an up to date state, and the recovering replica can find the data it needs. Many different protocols implement state transfer, and they have different ways of storing and transferring the state. The way these steps are done is key to increasing performance as much as possible. Replicas continue their usual operation while state transfers are happening, and those operations are often complex consensus algorithms with many communication rounds that need to finish as quickly as possible. Therefore, ensuring the transfer goes as smooth as possible is very important.

In the world of CFT systems, the Paxos consensus algorithm [10, 11] is one of the most famous implementations of SMR. It has initiated a great deal of research in the field, resulting in many different versions of SMR, both in CFT and BFT systems. The first implementation of a BFT SMR algorithm that was usable in practice was the *Practical Byzantine Fault Tolerance (PBFT)* algorithm [4]. PBFT provides a solution to the SMR protocol, along with a recovery mechanism explaining how to deal with failures. This mechanism consists of storing the state with the help of checkpoints and Merkle trees, as well as a clever way of asking for the state in order to avoid all replicas sending the whole state; this would create massive congestion in the network. Another novel approach that improves on the performance of state transfer is the *Collaborative State Transfer (CST)* done in BFT-SMaRt [2]. Apart from upgrading the logging and checkpointing schemes from previous works, CST improves upon the state transfer protocol by asking each replica for only a part of the state. Thus, cluttering of the network with unnecessary data is avoided, and other replicas do not waste as much time and resources on transferring the data. Most recent work [5] includes improving protocols such as CST even further, by taking geographical distance of replicas into account. This approach implements geographical SMR, so we will call this approach Geo-SMR. What happens here is that replicas are aware of the characteristics of each connection, thus adapting to the changes in the network by changing how much data they request from a certain replica. If a connection to a replica is fast, then the recovering replica will request more data from this one, compared to another replica with which it has a slower connection at that point in time.

2.2 Linux Traffic Control

Traffic Control (tc) represents a part of Linux which can be used to manipulate traffic coming into and going out of a machine. It is a name given to a plethora of commands involved in the following tasks surrounding network traffic: shaping, policing, scheduling, and dropping. Shaping

is concerned with controlling the rate of packet transmission; policing is the shaping of incoming traffic; scheduling can be used to rearrange incoming packets as needed; and dropping simply represents discarding packets. All of these are achieved with the help of three main objects: *queuing disciplines* (*qdiscs*), *classes*, and *filters*.

A qdisc is a scheduler to which the kernel enqueues the packets it wants to send to an interface. This scheduler proceeds to rearrange all those incoming packets according to its rules, and then sends them to the desired interface. The simplest qdisc is the FIFO qdisc. A qdisc can have classes, which give more flexibility and options when managing traffic. These classes can be used to prioritize traffic, or apply different rules to different destinations, depending on their address, port, etc. They can also contain other qdiscs, or child classes. These qdiscs are called classful qdiscs, and are the ones we will mostly be interested in. A qdisc which cannot contain a class is called a classless qdisc. Classes can also have multiple filters attached to them. Filters are what is used to actually enqueue a packet to a certain class. There are many available filters, however we will most notably use the one which allows us to filter based on packet information, such as destination IP address, destination port, etc.

One example of a classless qdisc is the *netem* qdisc [12], or the *Network Emulator*. This queuing discipline can be used to emulate a real world network and all its characteristics. We can add packet loss, duplication, delay, and even corruption to a network interface. This allows us to create a testing environment similar to large scale infrastructures, where we can test our new potential solutions in less time, and with less cost.

Traffic shaping is what we are most interested in, since we want more control over bandwidth allocation. For this purpose, we will focus mostly on the *Token Bucket Filter (tbf)* [18] and the *Hierarchy Token Bucket (htb)* [8] qdiscs, the difference between the two being `tbf` is classless, and `htb` is classful. With `tbf`, one can configure a maximum rate so that it is never exceeded, which is perfect in our case. There is a finite amount of bandwidth on one replica, and we do not want to go over it, even if its interface can handle more. However, since `tbf` is classless, we can only cap the bandwidth of the whole interface, which does not give us enough granularity and flexibility; ideally, we want to have different rates for different connections. This is where `htb` becomes immensely useful. Since `htb` is classful, we can create a class for each connection a replica has, and then change the maximum (*ceiling*) rate as the system evolves. Moreover, we can also set a minimum guaranteed rate for each connection, which means whatever is going on in the network, a replica is guaranteed to send the requested data at that rate at least. We can also add a priority to an `htb` class, thus telling the underlying system that, if there is excess bandwidth to be distributed, certain classes should get it first. To route packets to the correct class, we use filters based on whichever information we want, for example a destination IP address.

As can be seen from their names, both `tbf` and `htb` use a token bucket algorithm to achieve their goals, and they use the same one. This means that the traffic coming to these classes is filtered with the help of tokens. These tokens correspond approximately to bytes, and are refilled at a steady

pace. For each incoming packet, the amount of tokens corresponding to the size of the packet needs to be spent in order for the packet to be sent. If the tokens are unavailable, the packet is delayed until the bucket of tokens fills up again. The packets are dropped only once there are more packets waiting for tokens than the configured limit.

Chapter 3

Problem Analysis

There are two aspects of state transfer protocols which can potentially be improved upon. One is the ability of a system to handle replicas that are geographically distant, and we call this the *Receiver-Side Problem*, described in detail in the following sections. The other aspect is the reaction of replicas towards the dynamic nature of the network, i.e. are there better ways replicas can react to the changes in the network. This we call the *Sender-Side Problem*, also described in the following sections. For this particular project, we chose to focus on the latter problem, and find a solution for it.

3.1 Receiver-Side Problem

First, we discuss the receiver-side problem and how we designed our analysis. We implemented several different state transfer protocols: two naive approaches to state transfer and the Collaborative State Transfer (CST) from BFT-SMaRt [2]. The naive approaches serve as a good baseline to showcase how CST performs better. However, we also test CST under different setups, simulating geographically distant replicas, and this is where we some potential issues arise.

3.1.1 Design of the State Transfer Protocols

The first naive approach (*Naive1*) is a protocol where the recovering replica simply asks for the state in a sequential manner, going from replica to replica. It asks for the whole state immediately, so if the first replica is honest, the transfer is successful, and if not, the recovering replica goes on and asks the next replica. The second naive approach (*Naive2*) differs only in the fact that we ask multiple replicas in parallel. The CST protocol improves upon this by splitting the request into multiple parts, asking each replica for a piece of the state, rather than the whole thing. This achieves better results,

as seen in the next section, and at first glance it seems there is not much more one can improve. However, testing CST with additional changes unveiled some issues.

The main question is what happens when replicas are geographically very distant. To some extent, this is answered in Geo-SMR [5], however we want to follow this solution and see if we can find more room for improvement. The first variation of CST is with traffic control, i.e. using `tc` to simulate far-away replicas. Specifically, we use `htb` and `netem` to emulate different latencies and connection bandwidths. The second variation is the same, but we add the idea from Geo-SMR to split the requests in a more clever way, taking into account which replica is close with a good connection, and which replica is far with worse bandwidth and latency. We split the replicas to simulate half of them being in Europe, and the other half being in Japan, i.e. somewhere far in East Asia.

All of these protocols were tested in a system with four replicas, and in two different setups: the *All Healthy* case, where all replicas are honest and not crashing, and the *Healthy Byzantine* case, where none are crashing, but one replica is Byzantine, and is sending garbage data. We decided not to implement the communication rounds needed for hashes of the data and similar steps of all BFT state transfer protocols. This decision was made due to our desire to focus solely on the large file transfer; these other messages are usually very small in size and thus an insignificant contribution to the results. For this reason as well, all protocols persist the state in the same way: a write-ahead log that is periodically written to disk.

3.1.2 Test Results

The test results for these different implementations of state transfer can be seen in Table 3.1. The results shown in this table represent the average time it took to finish a 2GB state transfer under different protocols and setups, presented in seconds. The behaviour of the first three protocols was as expected, with CST performing better than the naive approaches, which were essentially similar in their results. The main difference between the two naive approaches is really the load on the network, the Naive1 approach being better in that regard, since fewer replicas are being asked for data. The interesting results lie in the last two protocols, i.e. last two columns of our table.

Evaluation of state transfer protocols for a 2GB transfer (in seconds)					
Protocol / Case	Naive1	Naive2	CST	CST with `tc`	CST with Geo
All Healthy	8.68s	9.94s	6.78s	50.09s	27.1s
Healthy Byzantine (close replica | far replica)	10.95s	14.79s	9.06s	88.9s | 51.44s	102.15s | 25.78s

Table 3.1: Evaluation of different state transfer protocols

When comparing CST with all replicas close to CST with the traffic control emulation, we can

see a drastic drop in performance. This was expected to some extent, and that is why there is the Geo-SMR approach. We can see that this version of CST cuts the transfer time in half, compared to CST that is oblivious to the geographical distance of replicas. However, what we noticed is that the time it takes for a transfer to complete can increase even further when one replica is Byzantine, particularly if that replica is close to the recovering one. In our environment, a recovering only has one close replica, therefore if that one is Byzantine, it has to ask for data from those far away. This creates a jump, on average, from 50 seconds to 89, or even worse from 27 seconds to 102 in the case where distance is actually taken into account.

This is the main conclusion gathered from these tests: even though the geographic distance is taken into account in existing solutions, there is still an issue of handling situations where a certain set of replicas is Byzantine, such as all those close to the recovering replica. In these circumstances, Byzantine replicas can severely damage a system's performance by significantly prolonging recovery. One immediate solution to this is making sure each region (let's say in this case continent, although one can define a region arbitrarily) contains at least $f + 1$ replicas. This way, there will always be a nearby honest replica from which to request data. However, there is a flaw to this solution, as it leads to designing systems with a lower tolerance for Byzantine replicas due to the f value being more expensive. Due to the existence of some work on replica placement in large-scale systems [13], we decided to focus on the Sender-Side problem in this project.

3.2 Sender-Side Problem

In Geo-SMR, we have seen how taking the dynamic nature of the network into account can lead to better results. However, this is only done by replicas that request a state transfer. Therefore, replicas providing the data act basically like a server, providing answers to requests when needed. The question here is, if senders take more information into account, can they improve the time it takes to finish a state transfer, and also their resiliency against Byzantine replicas. To answer this, we analyzed how Linux allocates bandwidth to its connections, what happens when multiple connections are active simultaneously, and how this can be altered with the use of traffic control.

3.2.1 Design of the Bandwidth Behaviour Tests

Firstly, we show how bandwidth behaves regularly, i.e. without any changes such as traffic control, and how it behaves when using `tbf` to limit the maximum bandwidth of the sender's interface. With two simultaneous receivers and one sender, the bandwidth is split evenly among connections, and when one closes, the newly available bandwidth is given to the other active connections. This confirms our assumptions that the kernel gives an equal share of bandwidth to its active connections. Since `tbf` does not give us more control, other than just limiting the speed of the interface, we use `htb` in order to get more granular control. Using `htb`, we devise a couple of scenarios and `tc` setups

to gain further insight.

All our scenarios have one sender and two receivers. One receiver has a connection with a speed of 1Gbps, and the other is a bit slower at 700Mbps. In the first scenario, a replica is trying to recover every 15 seconds, therefore sending a request to the sender. The sender acts as a server, and just starts sending data every time a request comes its way. The second scenarios sees the addition of the other replica, which is constantly asking for a transfer, i.e. being an annoying and noisy neighbour. As soon as the noisy replica finishes a transfer, a new one is initiated. This way, we test how a potential Byzantine replica (or just a very slow honest replica) can impact the performance of the system, and hinder other replicas in their recovery.

We evaluated these scenarios under two different `tc` setups. In both we use `htb`, however the difference lies in the minimum guaranteed bandwidth of the connections. The first setup does not set any guarantees, while the other sets these rates proportional to the ceiling rates the connections have. In our case, the faster replica would have 588Mbps minimum guaranteed rate and a 1Gbps ceiling rate, while the other replica, respectively, has 412Mbps and 700Mbps. The latter setup is what we call *`htb` with static allocation* throughout this thesis.

One might notice that the total sum of ceiling rates in these setups is over 1Gbps, which we set as our maximum interface speed. A natural question would be why do we allow network congestion to happen, when we can just set ceiling rates such that their sum equals our interface maximum. The answer is straightforward: if we do this, then a replica recovering alone will get a slower transfer than its full potential, since we had to set a ceiling rate lower than the connections actual full capacity. This is inefficient, and something we want to avoid.

3.2.2 Test Results

We measured the performance of both replicas in these scenarios and setups, i.e. both were in the role of the recovering and the noisy replica. The results can be seen in Table 3.2. "Scenario 1 - slower" indicates measuring the slower replica in scenario 1, and the same goes for scenario 2.

Evaluation of the two scenarios and their bandwidth for a 2GB transfer				
Scenario / `tc` setup	Scenario 1	Scenario 2	Scenario 1 - slower	Scenario 2 - slower
`htb` - no minimum rate	963.24 Mbps	766.4 Mbps	688.01 Mbps	212.46 Mbps
`htb` - static allocation	958.72 Mbps	582.73 Mbps	685.62 Mbps	416.09 Mbps

Table 3.2: Evaluation of the two scenarios and their bandwidth

Looking first at the results of Scenario 1, there are no surprises here, since there is only one replica that is requesting a state transfer, thus there are no intrusions. In both cases (faster and slower replica), the replica got its data at the maximum possible rate for its connection, that is

around 1Gbps and 700Mbps, respectively. Where we actually anticipated problems was in Scenario 2, and we ran into some unexpected results.

In scenario 2, we notice the drop in performance. When setting no minimum rates, comparing to scenario 1, the faster replica is 20% slower, while the slower replica is 70% slower. This asymmetry was unexpected, and it seems that without setting minimum rates, `htb` favours the faster connections. When setting guaranteed rates, the performance drops are 39% for each replica. This amounts to worse performance for the faster replica, however it is overall much more fair, which is what we want. Nevertheless, a noisy neighbour can still do a fair amount of damage, therefore this problem needs addressing, and is the one we aim to solve in this thesis.

Chapter 4

Design

In this thesis, we propose a solution for the aforementioned sender-side problem, called *Adaptive State Transfer (AST)*. More specifically, we propose three slightly different versions of the idea, of which one proved to produce the best results, as can be seen in Section 5.

4.1 Adaptive State Transfer

The principal idea of our solution is for a sender replica to allocate bandwidth to its connections in a more clever way. This can be achieved by taking into account more information about the system. This information consists of two things: how the characteristic of the connections change over time, and tracking how much resources replicas have already consumed. Based on this data, we recalculate the ceiling rates of connections, and with that also the minimum guaranteed rates, and apply the changes through traffic control (`tc`) commands. AST was designed with TCP as the underlying communication protocol.

Describing AST in more detail, we first explain when and why we change the ceiling rate of a connection. A diagram of these decisions can be seen in Figure 4.1. In the beginning, a replica sets the ceiling rate of each connection to its interface maximum, which in our case is 1Gbps. When no other information has been gathered yet, a replica always tries to service others to the fullest. The state transfer is done in chunks of 100MB, and the sender waits for an acknowledgment message (ack) from the receiver for each of those chunks. Based on the number of acks received per second, the sender can estimate the current bandwidth of the connection, i.e. at what speed the receiver can deliver the data. We then set the ceiling rate of that replica to the estimated value, at the end of the transfer. This way, the sender adapts to how the network changes. We only change the rate, however, when two conditions are met: 1) if the estimated rate was less than 90% of the current ceiling rate of the connection in question, and 2) if no other replica was participating in a state transfer while

the one in question was receiving the state. Since messages and communication protocols have a lot of overheads, we cannot assume perfect and ideal use of the bandwidth, and hence we have the first condition in place. The second condition ensures that we follow network changes other than the congestion created from state transfers. If a replica is the only one active, and it is slower than we expected, that means some external factors influenced the network. In case the rate is as we expected it to be (i.e. 90% of the current ceiling rate or more), we try and increase the ceiling rate by 20% of the interface maximum. Maybe there is less congestion in the network, or a connection got better and can now handle data faster. In order to react to these changes as well, we increase the rate (if we can, i.e. not going over the interface maximum), so that the next state transfer to that replica has the chance to be even faster.

The moment we change any ceiling rate, we need to recalculate all the minimum guaranteed rates. A replica's minimum rate is set in the same proportion to the interface's maximum bandwidth as its ceiling rate is to the total sum of ceiling rates for all replicas. For example, if a replica has 1Gbps ceiling rate, and the total sum of ceiling rates is 5Gbps, it means that this replica will get 20% of the actual available bandwidth of the interface, in our case 200Mbps (20% of 1Gbps). Therefore, whenever we change a replica's ceiling rate, we need to recalibrate the minimum rates of all replicas.

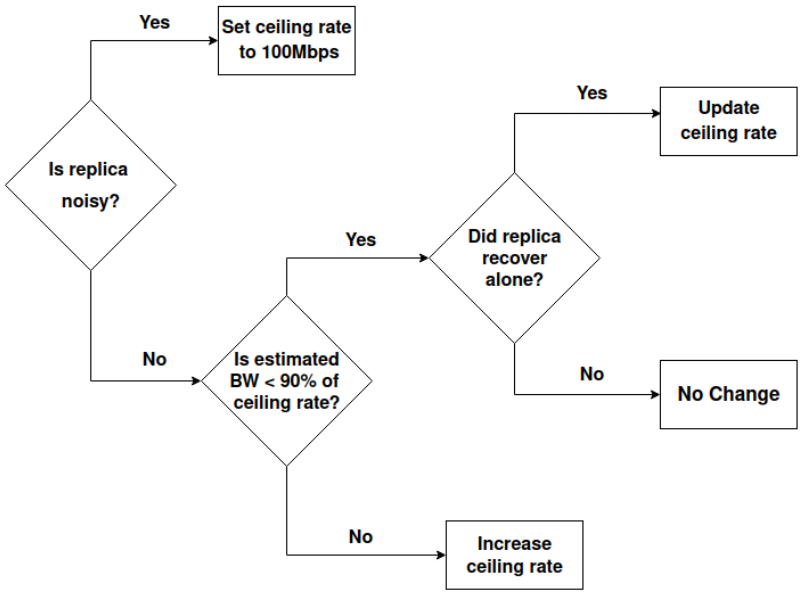

Figure 4.1: Diagram of the decision making process after a transfer is finished

Adapting these rates is one side of the issue. We also somehow have to track how "noisy" the replicas have been. We do this by defining a sliding time window, in which we keep a record of how much bandwidth each replica has consumed. We use the same estimation already calculated, and just add it to a sum we update each time a transfer for that replica completes. If a replica requests 4 or more transfers and has consumed 2500Mbps of bandwidth in the last 2 minutes, it is classified as

noisy, and its ceiling rate is set to 100Mbps. All of these values should, of course, be tailored to the system and application within which this algorithm is used. This would depend on factors such as how much bandwidth can be allocated to the state transfer protocol, or how frequent the transfers are expected to be, etc.

Based on the total bandwidth consumed within the time window, we decide each replica's priority. A replica with the lowest bandwidth consumed at that time gets the highest priority, which is priority 1. The next one gets priority 2, and so on. What we do with these priority values is simple. We reduce the newly calculated minimum guaranteed rates based on these priorities. For example, a replica with priority 2 has its minimum rate halved, a replica with priority 4 has its rate divided by 4, etc. Thus, we ensure that the rates are scaled according to the current priority. After this, we end up with some excess bandwidth that needs redistributing, and here is where we have a few different versions of our solution.

After all of these priority changes and recalculations, we simply create the necessary `tc` commands, and execute them, ensuring that our alterations are passed on to the Linux kernel.

4.1.1 Uniform Bandwidth Allocation

In the first, and original, version of our algorithm, we redistribute the extra bandwidth uniformly. Each replica gets an even part of that leftover sum of bandwidth, e.g. if there is 400Mbps leftover for 4 replicas, each gets 100Mbps. This can cause an overflow in the sense that the minimum rate becomes larger than the ceiling rate. We, of course, do not let that happen and in those cases just make the minimum rate equal the ceiling rate and continue with redistributing the excess in the same manner.

4.1.2 Proportional Bandwidth Allocation

The second version takes into account how much guaranteed rate each replica already has (after the priority-based reduction previously described). Whatever proportion a replica's minimum rate holds in relation to the interface maximum, that is the percentage of extra bandwidth that is allocated to it. For example, if the minimum rate is 20% of the interface max, then we add 20% of the excess bandwidth. In this version we are also careful not to create a guaranteed rate larger than the ceiling rate, similar to the first version.

4.1.3 Kernel-based Bandwidth Allocation

In the final version, we do not redistribute the excess bandwidth. Instead, we pass this work onto the kernel, or, to be more precise, `htb`. All we do is use our calculated priorities and put them in the `tc`

command. We use `htb`'s `prio` option to achieve this, and give each class (i.e. connection) a priority value. We already know that `htb` gives all excess bandwidth to the active connection with the highest priority. Although we did not like that as much, and wanted to create a more fair redistribution, we were still curious how this version will perform, and whether or not our solutions will outperform this one.

4.2 Influence of Byzantine Replicas

As we are concerned with BFT systems, keeping in mind what Byzantine replicas can do in our Adaptive State Transfer is very important. Since we added a little bit of new message passing in the form of acks, they could potentially be used against the system. Indeed, a Byzantine replica can lie and send ack messages as it wishes, making it seem very fast, very slow, or anything in between. Let us consider what can happen if a replica wants to lie and present itself as very quick. It cannot appear faster than the maximum rate at which the sender is actually sending the data. It is impossible for a replica to receive data at 1Gbps if it was sent at 500Mbps for example. Moreover, as the Byzantine replica appears fast, the sender will take note of it, and the replica in question would lose priority fairly quickly, or even be classified as noisy. Therefore, over time there is not a lot a Byzantine replica could gain from this.

On the other hand, a replica can appear to be extremely slow. At first glance, this might seem harmless; why would a Byzantine replica want to act as if it had a terrible connection? In AST, this can be a way for a Byzantine replica to ensure it has priority over all other connections. Over time, it can appear as it consumed very little bandwidth. To achieve this, the Byzantine replica can also be quiet and not ask for a state transfer for some time, before starting to bombard others with requests. Our two-minute sliding window mitigates this to a certain degree. We do not take all values since the beginning of the system into account, only those in the recent past. Therefore, a Byzantine replica trying to flood another replica with requests will either get classified as noisy, or lose priority fairly quickly, thus reducing its harm of the overall system.

Chapter 5

Evaluation

All of our designed solutions were first tested locally, however, in order to get relevant results, we set up some EC2 instances on AWS. Specifically, we used the c7i.8xlarge instances with Ubuntu Server 24.04 LTS as the operating system. We did not need a lot of disk space for our tests, so we used the default 8GB; what was more important was the network bandwidth. The reason we chose these instances, and not any weaker, cheaper ones, is the fact that they offer a stable maximum throughput of 12.5 Gbps. The smaller instances usually handle *up to* 12.5 Gbps, which means that if AWS servers are used extensively, the maximum available bandwidth would decrease and become unpredictable.

All implementations were done in Java, with openjdk 17.0.3, on Ubuntu 22.04.5 LTS. The reason for choosing Java is simply that it was the most familiar language to work with. Due to the time constraints of this project, we decided it was best to use the language we are most comfortable with. For this purpose of multi-threading and implementing distributed systems, Golang or Rust might have been better options, as they could lead to even better results. While evaluating our implementations, we used 2GB as the size of the state that is being transferred. State transfer in real-world systems are transfers of large amounts of data, larger than the 2GB we used. However, we needed to strike a balance between using large enough data to generate relevant results, and the time it will take to run all necessary tests and evaluations. Also, as explained previously, we limit the maximum possible speed of a replica's network interface to 1Gbps, since in real-world systems, a lot of bandwidth would be used by complex consensus algorithms, and it is realistic that a replica could spare only 1Gbps for state transfer.

5.1 Adaptive State Transfer with a Noisy Neighbour

First, we evaluate how our new solution works in the problematic Scenario 2 we defined; the decrease in performance which happens when there is an annoying, noisy replica constantly asking for a state transfer. We show how different versions of our algorithm compare, on average, to the naive solution that only uses `htb` to statically allocate the minimum and ceiling rates of each connection. We use this as the baseline, since it is the simplest way to try and solve this problem. These results are shown in Figure 5.1. The blue bars indicate different versions of our solution, and they are being compared to the orange bar in the graph.

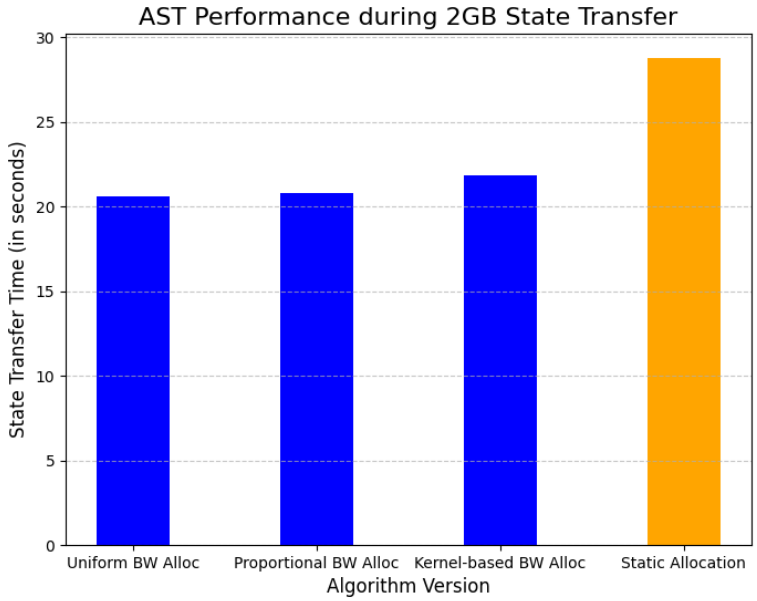

Figure 5.1: Average time of a 2GB transfer under different algorithms, with a noisy neighbour

From the values in Figure 5.1, we can conclude that our versions bring, on average, a faster state transfer, with the uniform allocation of bandwidth being the quickest. Our best result reduces the state transfer time by 28%. At first glance, the differences between these approaches are small, however there is an important detail to take into account: the lowest possible time it takes for a state transfer of this size over a network at a speed of 1Gbps is, on average, 17.4 seconds, which can be seen in more detail in Section 5.2. Therefore, there is only so much room for improvement; more specifically, 11.4 seconds on average. With our best version of the solution, we were able to close this gap by 72%.

The graphs in Figure 5.2 show a more detailed behaviour of our solutions. The line corresponding to "Theoretical Maximum" refers to the throughput which could be achieved in the ideal case, i.e. 1Gbps. The "Practical Maximum" presents the transfer over a network which has various

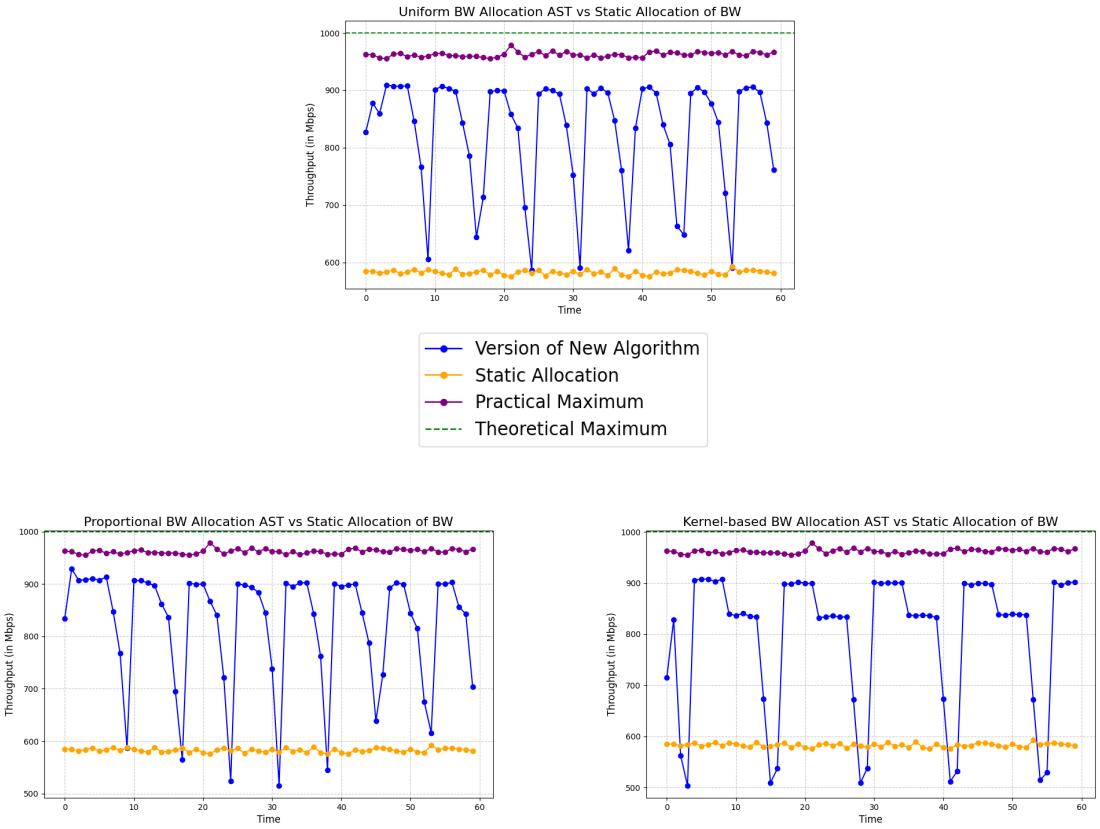

Figure 5.2: Detailed comparison of AST and `htb` with static allocation

communication overheads. This practical maximum is what we are ideally aiming towards for each state transfer.

The first rather obvious conclusion was that our algorithms behave in a much more chaotic fashion over time, compared to the one in which we statically allocate bandwidth with `htb`. The graphs in the figure correspond to the algorithm version as follows: the top one is the version with uniform bandwidth allocation, on the bottom left we have the proportional bandwidth allocation, and the bottom right is the third version, the one with kernel-based bandwidth allocation. Thanks to these results, we are able to see that giving too much priority (and with that bandwidth) to replicas in need can result in worse performance over time. This is due to the fact that, if they get a lot of bandwidth, they will eventually lose priority, as their overall consumption increases. This can be best seen in the third version of our algorithm, where we let `htb` handle priorities, and get the worst performance. We even see a fair amount of transfers achieve less throughput than with static allocation. `htb` always gives all excess bandwidth to the connection with the most priority, thus eventually giving priority to the noisy neighbour, and slowing down the replica trying to recover. We also see a bit of this behaviour in the proportional-allocation version. Although this version gives

marginally worse performance on average, throughput can also fall below what is achieved with static allocation.

Our original idea proved to be the best, although still giving a bit sporadic results. From these detailed results, we can say that the key to a good solution for fighting against a noisy neighbour is finding the balance in the allocation of bandwidth. If a replica has priority, we need to give it more bandwidth, or it will eventually starve. However, when doing so too much, the replica in question loses its priority, resulting in a larger number of longer transfers overall. Therefore, what is best is to find a solution which has a good balance; ideally keeping the throughput of all transfers above, and far from, the throughput we can get from the static allocation solution, while keeping most of them as close to the maximum as possible.

5.2 Adaptive State Transfer with One Recovering Replica

Although testing our solutions against problematic scenarios, such as having a noisy neighbour, is important, it is also crucial that our solution works well in cases where there is no contention for resources. It is expected that when a replica is the only one requesting a state transfer, it gets the maximum possible bandwidth that its connection can handle. We can see these measurements in Figure 5.3. Our solutions all perform as good as our baseline (again presented in orange colour), with them being slightly slower due to the implementation of our solution. Since the sender waits for acks for each chunk it sent, and carries out additional calculations, the overall performance does have a slight drop. The difference is less than a second on average for each version. The important takeaway here is that our algorithm does not create problems in cases where there never were any to begin with.

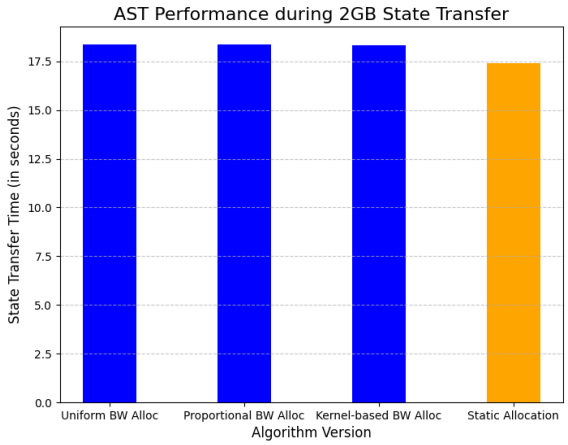

Figure 5.3: Average time of a 2GB transfer under different algorithms, with only one active replica

5.3 Adaptive State Transfer with Two Recovering Replicas

Up until now, we have discussed situations where there is either one replica alone that is trying to recover, or when that replica is hindered by a noisy neighbour, either a Byzantine node, or simply a slow one. Another relevant case worth testing is what happens when two honest replicas are trying to recover, almost at the same time. In this case, none of the replicas is Byzantine, or slow and noisy; they are simply two replicas that need to recover, and one of them starts recovering a bit earlier, before the other joins. The connection setup is as follows: one replica has 1Gbps ceiling rate, and the other has 700Mbps. One replica starts the transfer first, and after 5 seconds the second one follows. We evaluated both the faster one being first to initiate a transfer, and vice versa. When the slower one is first, we call these cases "flipped". We compare this to our, by now standard, baseline, which is the static allocation of bandwidth. For these evaluations, we used the original version of our solution, the uniform allocation of bandwidth, as it yielded the best results out of the three in the evaluation with a noisy neighbour.

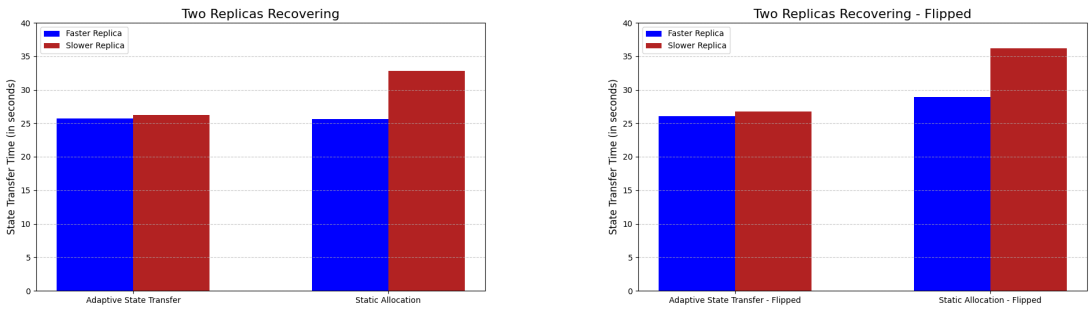

Figure 5.4: Algorithm comparison when two replicas are recovering simultaneously

The comparison can be seen in Figure 5.4. First, we can see that the performance of the slower replica is much better compared to only using static allocation, and that is in both regular and flipped cases. In the first case, the state transfer is 20% faster, while in the latter case it is 26% faster. Considering the ceiling rate of the slower replica, the fastest time for it to finish a transfer is, on average, 24.4 seconds. Therefore, there are gaps of 8.4 seconds and 11.76 seconds that can be closed, respectively in the regular and flipped cases. Our solution manages to close these gaps by close to 80% in both cases. In the regular case, the speed at which the faster replica finishes the transfer is very similar to our baseline, which seems unexpected. However, this is due to the algorithm ensuring more fairness, as discussed later, and as seen in more detail in Figure 5.6. In the flipped case of static allocation, both replicas are slower because they are in contention for a longer period of time, thus increasing both recovery times. Here, there is a slight improvement in the faster replica as well. We see here that static allocation does not react at all to what is happening in the system, unlike the presented algorithm. Our solution aims towards fairness for every replica in the system, and therefore a fair share of bandwidth to all replicas simultaneously requesting a state transfer. This

fairness should be, of course, proportional to each connection's capacity. This is further cemented in Figures 5.5 and 5.6, a more detailed look at the evolution of the system over time in this scenario.

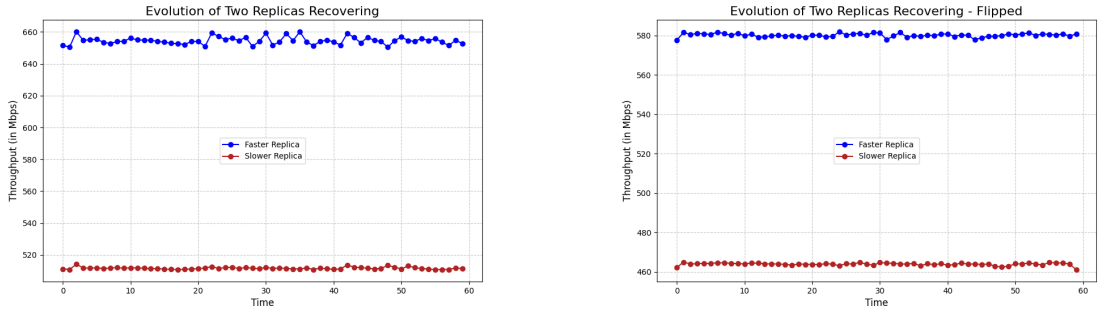

Figure 5.5: Static Allocation evolution when two replicas are recovering

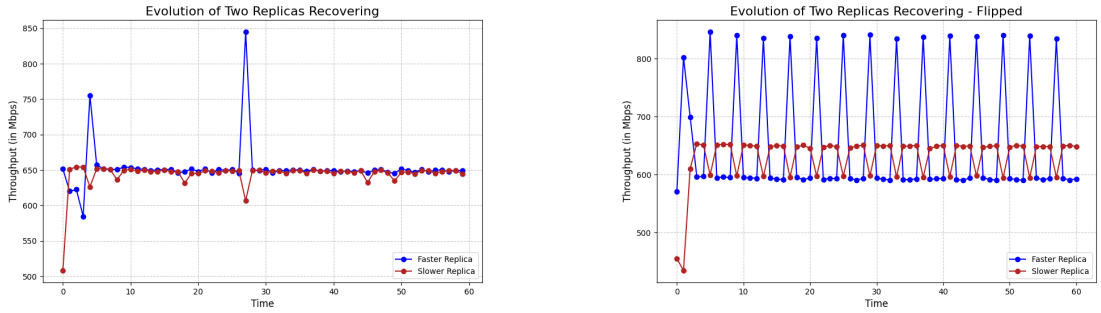

Figure 5.6: New Algorithm evolution when two replicas are recovering

Figure 5.5 confirms that static allocation will never take the past into account. A faster replica will always be faster than the slower one, regardless of the fact that it used a lot of resources during its lifetime. This seems rather unfair towards slower replicas, and how our solution improves on this is seen in Figure 5.6. Although there are obvious differences in the evolution of the system between the regular and flipped cases, both present an improvement on static allocation. In the regular case, the faster replica is first to initiate the transfer and, therefore, it has periods where it is the only one active and achieving throughput equal to its ceiling rate. Thus, it consumes more bandwidth and, most of the time, does not have priority over the slower replica. We see that the slower replica achieves close to optimal throughput in most cases, and the faster one gets the rest of the available bandwidth. If the slower replica had a different ceiling rate, e.g. 200Mbps, the result would follow the same pattern. The flipped case shows a different behaviour over time. Since the slower replica is first to initiate a transfer, it now gets the period when it is alone and getting optimal throughput. Therefore, it loses priority more often than in the regular case, and the faster replica gets a chance to achieve high throughput. Overall, both figures show a much better performance than we see on Figure 5.5.

Chapter 6

Related Work

Significant research has been conducted on SMR, with various different solutions, such as PBFT [4], BFT-SMaRt [2], and more recently Chop-Chop [3]. While their main contribution is a general SMR protocol, PBFT and BFT-SMaRt provide a detailed explanation on how they solve the problem of state transfer. They both introduce a novel idea of how to structure the state, in order to efficiently read and transfer it when needed. On the other hand, the work on Geographical SMR [5] also tackles the problem of bandwidth allocation, i.e. how to improve the use of bandwidth across the system in order to transfer the state as quickly and efficiently as possible. Due to their nature, these systems are expected to have servers across the globe, and thus the state of all connections can vary drastically. Although our work discusses a similar problem, the main difference lies in the fact that current research provides a solution from the receiver side, while we focus on the sender side. That work was the main motivation and inspiration behind this thesis, giving us ideas on where to go next in improving state transfer protocols.

Since we want to improve the use of the available (and a very finite quantity of) bandwidth, one might say we need a clever way of "scheduling" all connections when they want to get this resource. Therefore, solutions already implemented in scheduling CPU tasks, where there is a finite resource which a lot of entities require, seem like they should work just fine. The Completely Fair Scheduler (CFS) [15], currently widely used in Linux distributions, was a good starting point, while the brand new Linux scheduler, called the EEVDF scheduler [7, 17], provided even more insights and sparked ideas about how one can solve such problems. However, since bandwidth and CPU time are different concepts in the end, we cannot use these solutions as a black-box, but nevertheless some of these ideas could be used to further optimize bandwidth allocation among many connections.

Another aspect of the efficient state transfer protocol is how the state is stored on each machine. This is also something heavily explored already [2, 4, 20], however we opted to focus on the file transfer part of the state transfer protocol. Nevertheless, this aspect of the entire problem is very much relevant and extremely important in producing optimal solutions in the future.

Chapter 7

Conclusion

In conclusion, the solution we presented in this thesis improves on the current state transfer protocols by reducing the overall time it takes to successfully transfer files over the network. It does so by leveraging tools from Linux traffic control, and keeping track of what is going on in the system in real time. With this new algorithm, we were able to significantly reduce the time it takes for a replica to complete a state transfer in various scenarios where contention for resources is inevitable. Our work shows how replicas can manage their available bandwidth, in order to ensure fairness and optimal performance during simultaneous state transfers. Although similar work already exists on the receiver side of state transfers, we crucially show that improvements can also be made on the sender side.

7.1 Future Work

The solution we presented here is a rather simple one, however it already yielded positive results. Due to the time constraints of this project, we were unable to implement all the ideas we had. Therefore, there are a few directions in which this research can be continued in the future. The way we track how annoying a replica is can be more complex and take even more information into account, such as the size of the requests. Also, as we had multiple versions of redistributing excess bandwidth, it is possible that there are more efficient versions to be designed. Moreover, testing these solutions with full implementations of consensus algorithms, and alongside the most recent state transfer algorithms, would provide solid insights into how well this solution works, and how it could be further improved. Moreover, in this project we solely focused on the file transfer part of the whole protocol. To accompany this, further research could be done to improve how replicas store their state, and how the data is transferred from the disk to the kernel, before it is sent via the network interface.

Bibliography

[1] Werner Almesberger et al. *Linux network traffic control—implementation overview*. 1999.

[2] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. "On the {Efficiency} of Durable State Machine Replication". In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 169–180.

[3] Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. "Chop Chop: Byzantine Atomic Broadcast to the Network Limit". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 269–287. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/camaioni.

[4] Miguel Castro and Barbara Liskov. "Practical byzantine fault tolerance and proactive recovery". In: *ACM Transactions on Computer Systems (TOCS)* 20.4 (2002), pp. 398–461.

[5] Tairi Chiba, Ren Ohmura, and Junya Nakamura. "A State Transfer Method That Adapts to Network Bandwidth Variations in Geographic State Machine Replication". In: *2021 Ninth International Symposium on Computing and Networking (CANDAR)*. IEEE. 2021, pp. 88–94.

[6] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. "Atomic broadcast: From simple message diffusion to Byzantine agreement". In: *Information and Computation* 118.1 (1995), pp. 158–179.

[7] *EEVDF Scheduler - The Linux Kernel Documentation*. https://docs.kernel.org/scheduler/sched-eevdf.html. Last Accessed: 2025-01-07.

[8] *htb: Linux man page*. https://man7.org/linux/man-pages/man8/tc-htb.8.html. Last Accessed: 2025-01-10.

[9] Bert Hubert et al. "Linux advanced routing & traffic control HOWTO". In: *Netherlabs BV* 1 (2002), pp. 99–107.

[10] Leslie Lamport. "Paxos made simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58.

[11] Leslie Lamport. "The part-time parliament". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 277–317.

[12] *netem: Linux man page*. https://man7.org/linux/man-pages/man8/tc-netem.8.html. Last Accessed: 2025-01-10.

[13] Shota Numakura, Junya Nakamura, and Ren Ohmura. "Evaluation and ranking of replica deployments in geographic state machine replication". In: *2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE. 2019, pp. 37–42.

[14] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 305–319.

[15] Chandandeep Singh Pabla. "Completely fair scheduler". In: *Linux Journal* 2009.184 (2009), p. 4.

[16] Fred B Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[17] Ion Stoica and Hussein Abdel-Wahab. "Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation". In: *Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22* (1995).

[18] *tbf: Linux man page*. https://man7.org/linux/man-pages/man8/tc-tbf.8.html. Last Accessed: 2025-01-10.

[19] *tc: Linux man page*. https://man7.org/linux/man-pages/man8/tc.8.html. Last Accessed: 2025-01-10.

[20] Léonard Vaney. "State persistence in state machine replication context". In: *Student semester project at the DCL lab at EPFL* (2024).

[21] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: BFT consensus with linearity and responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 347–356.